Certification Authorities Software Team (CAST)

Position Paper CAST-20

ADDRESSING CACHE IN AIRBORNE SYSTEMS AND EQUIPMENT

COMPLETED JUNE 2003

(Rev 1)

NOTE: **This position paper has been coordinated among the software specialists of certification authorities from the United States, Europe, and Canada. However, it does not constitute official policy or guidance from any of the authorities. This document is provided for educational and informational purposes only and should be discussed with the appropriate certification authority when considering for actual projects.**

Addressing Cache in Airborne Systems and Equipment

1.0 Introduction

Many applicants utilize cache memory in microprocessors to accelerate the speed of memory access, thus improving processor throughput. However, using cache memory increases the complexity and accuracy of the worst-case execution time (WCET) analysis and introduces challenges for coherency, deterministic execution, correct memory management, and partitioning protection. This paper considers some of the concerns and current approaches to addressing cache memory usage in airborne systems and equipment.

2.0 What is Cache Memory and Why is It Used?

Cache memory is a relatively small and fast memory used as temporary storage for data and instructions by the central processing unit (CPU) of modern microprocessors. A memory manager is typically used to optimize the contents of the cache memory for maximum throughput.

Modern processors are increasingly reliant on cache memory. Some processors even have two levels of cache memory: L1, which is primary cache (small and fast with no wait states) and L2, which is secondary cache (slower than primary cache since it has some wait states, but faster than main memory) [2].

3.0 What Are The Certification Concerns with Cache Memory?

There are number of concerns regarding the usage of cache memory. In many airborne systems containing multiple software functions of different software levels, cache memory is a common resource used by all the software functions executing on that microprocessor. Therefore, providing protection mechanisms and partitioning between those functions can be very challenging and necessitates thorough analyses and complete robustness testing of the approach and implementation of cache memory management. Additionally, many cache management approaches rely heavily on commercial-off-the-shelf (COTS) hardware components, such as memory management units (MMUs) and watchdog timers, that, if relatively new and untested, may not have a history of reliable and predictable performance integrity. In many safety-critical real-time systems, certain "critical" code functions must execute at a specific frequency (or when certain events occur) reliably, while other software functions in the same application may have less time-critical need for control of the processor and its resources. Obviously, safety-related time-critical code must execute reliably and timely, independent of other function's (sharing the processor resources) needs. Other concerns revolve around WCET analysis complexity and accuracy when using cache memory. Some of the specific certification concerns are described below (not an exhaustive list):

3.1 Establishing An Accurate WCET

DO-248B/ED-94B, FAQ #73 includes the following text [4]:

DO-178B/ED-12B[1] states that the worst-case timing should be determined. Section 6.3.4f of DO-178B/ED-12B states that as part of meeting the verification objective of the source code being accurate and consistent, the worst-case timing should be determined by review and analysis for Levels A, B, and C software. The results of this review and analysis should be documented in the Software Accomplishment Summary as timing margins (reference Section 11.20d of DO-178B/ED-12B).

The worst-case timing could be calculated by review and analysis of the source code and architecture, but compiler and processor behavior and its impact also should be addressed. Timing measurements by themselves cannot be used without an analysis demonstrating that the worst-case timing would be achieved, but processor behavior (e.g., cache performance) should be assessed. Using the times observed during test execution is sufficient, if it can be demonstrated that the test provides worst-case execution time.

Currently, one of the major concerns with fully cached systems is the inability to establish a predictable timing evaluation. The concern is not only with data cache but also with instruction caching (which includes pipelining and branch prediction algorithms). If the program has no interrupts, the concern might be trivial. However, when asynchronous interrupts are allowed, the path of execution of the program has no tracking of the cache state; therefore, the time at any given point in a program to fetch the next item from memory (data or instruction) is unknown. Figure 1 below shows how pipelines, caches, registers, and main memory are integrated in a typical modern processor. Figure 2 illustrates the pipeline structure for the PowerPC 603e. Both of these illustrations are borrowed from [2].

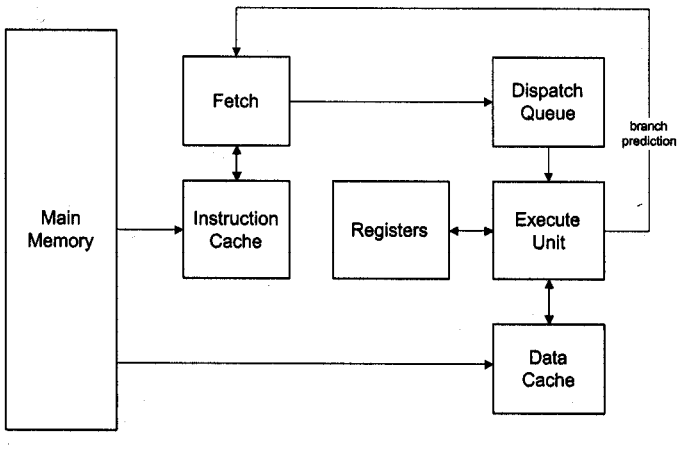

Figure 1 - Typical Processing Architecture

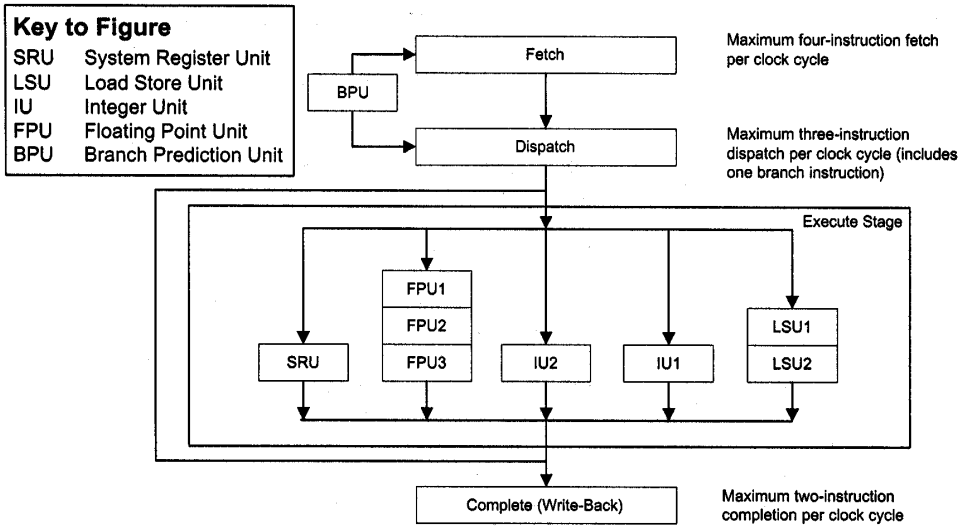

Figure 2 - Pipeline Structure of a Typical Modern Processor - The PowerPC

The predominant concern is the ability to predict the timing behavior of the next fetch or a given sequence of instructions and associated data execution paths.

3.2 Variations Execution Times

Use of cache memory results in greater variations in the execution times of the software [3]. Again, this makes accurate predictability a concern.

3.3 Cache Modeling Accuracy

The more complex the processor, the more difficult it is to validate the models used to predict cache performance. With memory cache enabled, it is more difficult to deduce the actual WCET of the software, because the analysis would need to include a model of the cache mechanism [2]. Since the cache management mechanism is typically COTS hardware, obtaining an accurate model is difficult or impossible. In some cases, it might be assumed that all memory accesses will result in a cache miss; however, this leads to wasted resources [2].

3.4 Real-Time Operating System Use of Cache

When real-time operating systems (RTOS) and run-time systems are used, they utilize the processor's cache memory.

The FAA recently sponsored research on COTS RTOS. One excerpt from the report states [5]: *Cache memory is indeed more of a hardware issue, however complex microprocessors that contain and use cache memory do have an affect on the performance of the RTOS. Under consideration is how should one handle non-deterministic cache that is shared among partitions. Complex cache algorithms may*

require a validation of the caching scheme for determinism and cache overhead should be accurately accounted.

A particular area of concern is partitioned RTOSs. The FAA report, section 4.1.3.4 states the following regarding cache memory and partitioning [5]:

An area that deserves particular attention is the use of cache memory in a partitioned computer platform environment. A cache is typically small-size, high-speed memory, or a hierarchical set of memory, that resides between the CPU and the main memory. Cache memory contains a copy of the most frequently accessed memory locations, which can reduce the overall memory access time. The use of cache can lead to non-deterministic execution time for functions, depending on how much of the data needed by the function is available in cache. This behavior may be further aggravated by the fact that cache is typically a shared resource among partitioned functions, which may lead to cross-interference among partitioned functions in the time domain, and violate the partitioning protection. Depending on the state in which the cache memory is left by a function in a partition, the execution time of the next function scheduled to execute may vary. Even though the execution time of a function is non-deterministic due to cache, it is still bounded by the worst case of having all accesses directly to/from main memory. Since worst-case analysis is crucial in safety-critical, real-time systems, timing analysis, and scheduling to tasks should address protection of the partitioned functions considering the presence and use of cache memory. Interference of cache in the spatial domain can be controlled using memory protection mechanisms such as MMU and SFI. However, the use of cache introduces an additional concern of maintaining cache coherency such that the cache swapping is valid at all times. Changing a datum only in cache or main memory, without reflecting it in its copied version, may result in inconsistent or erroneous behavior. Techniques for preserving cache coherency should be verified, and the overhead should be accurately analyzed and addressed in worst-case scenarios.

3.5 Cache Coherency

Maintaining coherency between the data cache and RAM and between the instruction and data caches is a concern when cache is used. Different architectures tend to use cache in a variety of ways, making it difficult to address with a single solution. For example, there are two separate caches on the PowerPC, one for instructions and one for data. Each cache has its own encoding to control the cache coherency. Figure 3 below shows potential coherency issues with the PowerPC (PPC).

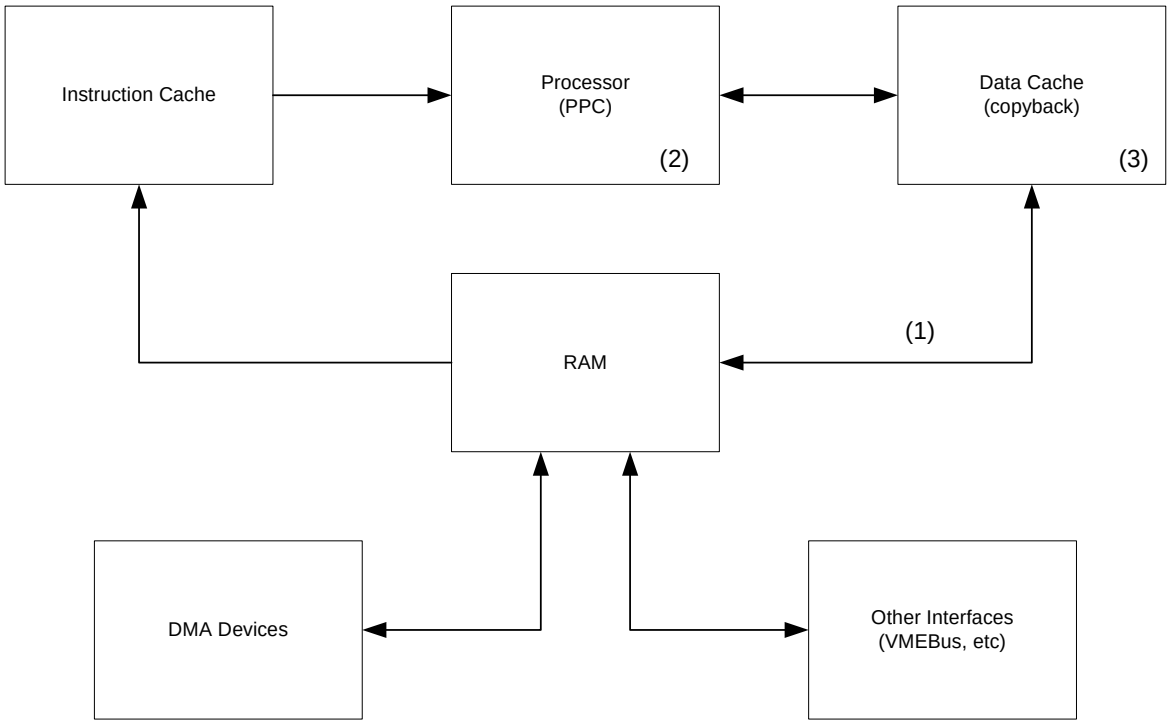

Figure 3 – Cache Coherency

The loss of cache coherency may occur in three places:

(1) Between data cache and RAM. This results from asynchronous accesses (reads and writes) to the RAM by the processor and other masters. Adding code into the drivers to address the other masters can often remedy this problem.
(2) Between instruction cache and data cache. This results in cache getting out of sync when the loader, debugger, or interrupt connection routines are used. Instructions may be loaded into data cache but not instruction cache, resulting in a coherency problem. This is typically addressed by flushing the data cache entries to RAM and invalidating the instruction cache entries (this will be addressed further in section 4).
(3) Shared cache lines. When cache lines are shared by more than one task, coherency problems may be introduced. This is often addressed by allocating memory on a cache line boundary, then rounding up to a multiple of the cache line size.

3.6 Cache Jitter and Partitioning

Section 2.2.6 of [7] introduces the concerns of caching on partitioning by stating:

Cache memory is a hardware architecture mechanism whose primary purpose is to improve performance of an application running on the target processor. It accomplishes this by holding information in a local high-speed cache memory, and synchronizing this information with the contents of main memory as needed. Cache memory should receive special scrutiny in a partitioned system because the cache mechanism is not aware of the address partitioning architecture. When an application runs within one partition, it forces the CPU to load information, which is then preserved for as long as possible

within the cache. Subsequent partitions may be affected by the presence of data in the cache. As cache memory is common to all partitions, the use of this resource requires careful analysis. Write-through and/or cache flushing techniques, among others, may be needed.

Section 4.2 of [7] specifically addresses cache jitter as it relates to partitioning:

Cache memory is a global resource that is shared between partitions. The information in cache may contain page address values, data values, and code. When a reference to a particular value is encountered, an automatic search is performed in the cache memory. If the value is found, it is used directly (cache hit); if the value is not found, it must be loaded (cache miss). The load is much slower, as it requires an off-chip memory access cycle. Once loaded, the value may be re-used for as long as it remains in cache memory. The cache is very fast but expensive (per bit), so it is much smaller than typical off-chip memories. See section 2.2.6 for details.

Imagine four partitions: P1, P2, P3 and P4. Assume a simple scenario where these partitions have a duration of 25 msec and a period of 100 msec. According to our rules for partitioning, it must be impossible for any partition to influence any other or to only affect other partitions in well-defined and controlled ways. During integration testing it is determined that all partitions complete their processing in their allotted time slots. This may have been accomplished because partitions P1, P2, and P3 perform some very intensive computations that use little data and very little code during the computations. Code can consist of small, highly computational loops.

Under these conditions, three partitions may use very little of the cache. Partition P4 will have much of the cache available to itself, and it will have very few cache misses in each iterative period. Under "normal" conditions, P4 will complete its work within its allotted period.

Imagine the situation where P3 suddenly changes its mode of operation, such that it references a large amount of data and executes a lot of code just once rather than referencing a small amount of code iteratively. P3 will cause P4 to have many cache misses for both code and data. This may result in P4 growing by a factor large enough to cause it to miss its deadline. Clearly, this violates the principles of partitioning, because one partition has caused another to fail, which is unacceptable.

The problem may be lessened by selecting different caching modes. In 'copy-back' mode, a data value is held in the cache until space is required for a new value. Before the cache memory location obtains its new value, the old value is written to the target memory location. The memory store and memory load are still performed every time a data value is updated, but several updates in memory may be performed without extraneous reads and writes. Absence of cache misses reduces the reads and copying back to memory.

A 'copy-through' cache policy forces the value in cache to be copied back to memory each time a value is updated. As long as a value is in cache, it will be re-used, but as soon as it is changed, it is written back to memory. This policy reduces the read from memory, but it does not decrease the writes.

It may appear that the 'copy-through' policy is always more efficient and should be used exclusively, but this may or may not be the case depending on the distribution of the data accesses.

A 'data value read' results in the placement of the value in the referenced memory location as well as the adjacent locations. Depending on cache-line length, this could be a 32-byte memory fetch and a corresponding 32-byte write. If a copy-through policy is used, then only one memory location is written. If memory reference density is high, then the copy-back policy is more efficient otherwise; the write-through may be more efficient, depending upon the program.

Data reads are not affected. Because code is not copied back, it is unaffected by the copy-back mode.

[7] goes on to say: *The loss of processing throughput due to cache misses becomes more important if it affects short time events. A partition may have processes that perform calculations, and which synchronize with other processes. These synchronization events may be issued with time outs so that if a process is blocked, its blocking time is limited. In the presence of delays caused by cache misses the calculation times, and thus synchronization responses, may be affected. Cache misses can affect the behavior of processes within a partition, even though the effect can be bounded when analyzed over a partition's duration.*

Design assurance concerns must include not only the cache-induced jitter on partition start and duration, but the effect on internal timing events as well.

3.7 Cache Failure

When a system relies on cache, the failure of that cache (e.g., loss or corruption of cache data due to power interrupt) should be considered in the system safety assessment.

4.0 Approaches to Dealing With Cache

A number of approaches have been or are being proposed regarding the use of memory cache. Some of those approaches are discussed below:

4.1 No-Cache Approach

One approach is to turn off cache or to use the no-cache as the data point for the WCET. The no-cache approach usually severely inhibits the processor performance and is not acceptable for most systems. (Note: When a system relies on cache, the failure of that cache should be considered in the system safety assessment. The no-cache approach may help to validate the safety assessment.)

4.2 Use Processor Tools

One approach is to use the various processor tools to manage cache in an intelligent way that guarantees good performance for the majority of the functions (e.g., all safety-critical or time-critical functions). The concern here is the tool accuracy, reliability, integrity, and determinism.

4.3 Statistical Evaluation

Another potential approach (suggested by some but not yet implemented) is to use some sort of statistical evaluation of the timing analysis and relate it to a safety analysis. Note that this is very close to a statistical problem that obeys the usual rules of randomness and some estimation of the probability of being with a given timing budget could be obtained and possibly entered into the safety analysis. There are several issues dealing with what inputs would qualify to ensure a truly statistical description of the timing but it may be a feasible approach.

4.4 Removing Signals

When there is a disparity in the severity of hazards between corrupted outputs and removal of outputs, another potential approach may be considered. For example, loss of an ILS signal can be considered to be minor, whereas misleading ILS could be considered hazardous. In these cases, detection of a timing problem could result in removal of the signal before it resulted in potentially misleading guidance.

4.5 System Architecture

For some systems it may be possible to architect the system in such a manner that it is impervious from a safety viewpoint due to timing variations. This clearly depends on the system and is not a universal solution.

4.6 Inserting Code

It might be feasible to insert "snooping code" into the microprocessor to determine its behavior and assist in accurate WCET analysis.

4.7 Addressing Cache Coherency

Some of the approaches for dealing with cache coherency were briefly mentioned in section 3.5 above. Addressing cache coherency in multiple architectures typically requires designing for the case with the greatest number of coherency issues (i.e., assume Harvard architecture (separate instruction and data caches), copyback mode, DMA devices, multiple bus masters, and no hardware coherency support). Typically, hardware is the most optimal way to maintain cache coherency.

4.8 Addressing Cache Jitter and Partitioning

As described in 3.6 above, cache jitter can affect partitioning assumptions. When considering design assurance, the applicant should address the following questions [7]:

- Does the behavior of the cache affect time allocated to a partition and performance?
- Can the worst-case partition execution time be tested, by flushing the cache in a preceding partition window?
- Is there a mechanism provided to control the cache behavior at partition switches?

- Does the system integrator control the cache behavior through the system configuration tables?

[7] states:

There are several solutions to make the cache behavior more predictable:

- *Switch the cache off: While this may seem attractive because it makes each of the partition execution profiles deterministic, it places a large performance penalty indiscriminately over all of the partitions.*

  Most programs assume the availability of cache memory and expect the performance improvement that they are given.

- *Specify a huge jitter margin: When designing the partitioned system, specify that each partition switch will include a margin that is equivalent to a cache miss on every value in the cache. This overhead is subtracted from each partition's duration. Each process in a partition must be verified independently under these timing conditions.*

  The disadvantage to this solution is that each partition is subject to the same fixed margin. A partition with a very short duration looses a large percentage of its permissible processing capability, especially if the cache is large.

- *Selective flushing: For those system applications that require very deterministic performance, the cache could be flushed during the partition switch such that the incoming partition has a clear cache memory at the start of its duration. Flushing means copying all of the cache values only present in the cache back to main memory (i.e., they have been updated and copy-back mode is used).*

  This places the overhead at the start of the partition rather than it being distributed throughout. The time taken to perform this operation is not fixed, as it depends on the number of values that must be written to memory.

- *Selective Invalidate: For system applications executed in a write-through mode for which deterministic execution is required, the cache may be invalidated during a partition switch. A cache invalidate is very fast. A single instruction makes the cache appear empty. Subsequent data reads are stored in the cache and reused as normal.*

The start of a partition can be precisely predicted. The timer interrupt to start the partition switch should be very accurate, and the time to switch context (save and restore all of the registers) will be constant. Additional overhead to select the next partition should also be small, and its maximum well specified, especially as the partition scheduling is based on a simple round-robin algorithm.

4.9 Cache Flushing

Cache flushing has been mentioned a few times in this paper. It is a practical approach that is frequently used in cached systems to preserve cache coherency and to deal with partitioning concerns. It prevents a partition from depending too much on the history (i.e., by flushing the cache, the current partition doesn't depend on previous partition

execution); there is also less "variability" in the execution time (as the cache memory state "starting point" is always the same at the beginning of the partition execution).

4.10 Other Approaches

[6] proposes that the following four technologies are most-often used for measuring software execution speed:

- Logic analyzers
- In-circuit emulators
- Hardware-assisted software performance monitors
- Software assisted software performance profiles

According to [6], the technologies typically use one or more of the following fundamental measurement methods:

- Determining where the system is spending its time (e.g., profiling).
- Monitoring the ability of critical sections of code to meet their deadlines.
- Measuring a system's response to external events.

All of these measurement techniques have some kind of effect on the system performance. Therefore, the performance measurement technique must take into account the effect on the software being developed.

5.0 Certification Authorities Position

The issues presented in section 3 of this paper should be addressed by applicants using cache, as well as any other "shared" processor resources and project-specific issues. Applicants should address any use of cache in their software plans. In some cases, a project-specific issue paper or certification review item may be needed to obtain agreement between the certification authority and applicant. Section 4 provides some potential approaches; however, the details of these approaches should be closely coordinated with the certification authorities. Most applicants use proprietary approaches that have not been detailed in this paper.

6.0 References

[1] RTCA/DO-178B and EUROCAE/ED-12B, "Software Considerations in Airborne Systems and Equipment Certification," 1992.

[2] "Use of Modern Processors in Safety Critical Applications." By Iain Bate, Philippa Conmy, Tim Kelly, and John McDermid (of University of York).

[3] Reference used in [2]. "Static Cache Simulations and Its Applications," by F. Mueller, Department of Computer Science, Florida State, PhD Thesis, 1994.

[4] RTCA/DO-248B and EUROCAE/ED-94B, Final Report for Clarification of DO-178B/ED-12B "Software Considerations in Airborne Systems and Equipment Certification," 2001.

[5] Study of Commercial-Off-The-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications. Research performed by UTRC under contract to FAA. Draft report submitted to FAA in May, 2002.

[6] "You Can't Control What You Can't Measure, OR Why It's Close to Impossible to Guarantee Real-Time Software Performance on a CPU with On-Chip Cache" by Nat Hillary and Ken Madsen. SE World, 2002.

[7] Commercial Off-The-Shelf (COTS) Real Time Operating Systems (RTOS) and Architectural Considerations. Draft FAA Report, Submitted May 15, 2003. (Note: Final report will be posted on http://av-info.faa.gov/software in the future.)